TU Clausthal

Clausthal University of Technology

An open agent architecture: Fundamentals

Peter Novák

IfI Technical Report Series **IfI-07-10**

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

An open agent architecture: Fundamentals

Peter Novák

Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany
peter.novak@tu-clausthal.de

Abstract

Different application domains require different knowledge representation techniques. Therefore a robust agent oriented programming framework should not commit to a single KR technology. Instead, it should facilitate an easy *integration of heterogeneous knowledge representation techniques* in a single agent system. Similarly, different situations an agent might happen to be in, require different schemes of behaviours, such as reactive vs. sequential. Mixing these can lead to a more robust agent system. Finally, a complete agent-oriented programming system should be complemented by a logic for reasoning about resulting implemented agent programs and ideally serving as a methodological framework as well.

In the first technical note of the series on *open agent architecture* we introduce theoretical fundaments of a novel agent programming framework of *Behavioural State Machines*. The presented framework draws a strict distinction between a knowledge representational and a behavioural level of an agent program. It 1) supports a high level of modularity w.r.t. employed KR technologies, 2) allows implementation of hybrid agents, i.e. mixing of both reactive, as well as sequential behaviours, and 3) provides a clear and concise semantics based on a well-studied computational model of Gurevich's *Abstract State Machines*.

1 Motivation

The paradigm of *agent and multi-agent systems* bridges the gap between cognitive robotics and programming intelligent software systems. In open multi-agent systems agents operate in highly dynamic and often unstructured environments with incomplete information about, and at best only a partial

control of it. In the last years we witnessed a stream of inspiration from cognitive robotics into the agent programming research community.

Nowadays the landscape of agent programming languages and frameworks is governed by BDI [15], one of the most influential architectures for *rational agents* [20]. It became a source of inspiration for development of several programming frameworks for software agents (for a survey of state-of-the-art see e.g. [3], or [4]). Most of these frameworks try to some extent implement axiomatic systems of either the original BDI architecture's *I-System* [15], or other similar formal theories of agent's mental attitudes (e.g. [7]). In the following we focus on a family of systems built from first principles and providing a clear formal semantics w.r.t. a formal model of BDI rational agents such as e.g. *AgentSpeak(L)*/*Jason* [14, 5].

A fundamental difficulty with a straightforward transfer of such axiomatizations to implemented agent-oriented programming frameworks appears to be an inflexibility w.r.t. agent's knowledge representation and a too tight coupling to a particular model of agent rationality. In the following, we substantiate our claims on the *I-System*, which is a set of postulates roughly axiomatizing the internal mechanics governing a BDI-based agent.

Informally, *I-System* specifies that an agent should adopt only goals it believes to be an option (AI1). It should adopt intentions in order to achieve its goals (AI2). If an agent has an intention to perform an action, it will eventually also perform it (AI3). It should be aware of the fact that it committed itself to certain goals and intentions (AI4, AI5). If an agent intends to achieve something, it also has to have a goal to intend it (AI6). It should be aware of its actions, i.e. if it performs an action, it should also believe that it performed it (AI7). And finally an agent should never hold its intentions forever, i.e. each intention must be eventually dropped (AI8).

In an attempt to implement elements of the *I-System* in a practical programming framework, some of the requirements impose strong constraints on the internal mechanics of implemented agents and their semantics. Because of the necessity to relate agent's mental attitudes, such as goals, beliefs and intentions, (AI1, AI4-7), most of the time programming frameworks bind to a single KR technology. Mostly, the KR of choice is *Prolog*, or a similar logical language. On the other hand, to comply with axioms of internal dynamics and interactions between agent's mental attitudes (AI2-3, AI8), authors introduce intricate semantic rules bound to a particular choice of attitudes and a model of rationality.

Obviously, such decisions make a theoretical study of implemented agent systems easier. A rigorous system semantics in terms of logic-based languages allows application of formal methods like formal system specification, model checking, or verification. However, such radical design decisions have also strong consequences on practical applicability of a particular agent programming framework. A limited support for integration of heterogeneous KR technologies in a single agent limits the applicability of the framework only

to domains, in which the chosen KR technology is appropriate. A choice of mental attitudes, a programming framework implements, and the nature of interactions between them, also prescribes a certain programming style, which might not always fit a particular domain. Finally, the above mentioned constraints, together with a relative semantic intricacy of a deliberation cycle provided by a particular framework dramatically raise the threshold for adoption of such a framework by casual programmers.

We take a more liberal, rather an *engineering* stance to the design of agent programming frameworks. The purpose of this paper is to propose an agent oriented programming framework devised from basic principles, driven mainly by a need for an *open, modular and pragmatic agent architecture*. I.e. one not dictating a programmer ways to design and implement his[1] application, yet allowing him to freely exploit techniques at hand, even if that would mean a bad practice. Rather than a strict language semantics, a system developer should be able to choose implementation techniques and an agent model suitable for a specific application domain. It should be rather a set of *methodological guidelines*, which lead a programmer to a design of a practical agent system, rather than a domain independent choice made by authors of a considered programming framework.

The result of the motivation analysis above are three main issues an agent architecture has to address: *knowledge representation modularity*, encoding of *agent system dynamics* and *programming language semantics*. After briefly discussing these principal issues in Section 2, in Section 3 we introduce a computational model of *Behavioural State Machines* (*BSM*) tackling them. Subsequently, in Section 4, we introduce *Jazzyk*, an implemented programming language and interpreter for *BSM* and discuss appropriateness of the *BSM* model for agent oriented programming on an example of an office space robot. In Section 5 we discuss relevant related work and finally Section 6 concludes the paper with an outlook to the on-going and future work.

2 Desiderata

Consider the general definition of an *agent* [20]: an agent is a *situated* and *embodied* software entity, which *autonomously* acts in its environment, *proactively* pursues its (or its user's) goals and *reacts* to changes of, and events in, its environment. Obviously, we omit here agent's social abilities, which we briefly discuss later in the Subsection 4.2. Additionally, we assume the agent's environment to be *dynamic* and *unstructured*. The agent builds a model, or a *representation* of it and uses it to infer its situation-specific behaviour(s).

[1] Without any gender prejudice, to ease readability, we always refer a 3rd person in masculine.

Behaviours Agent's task in its environment is to autonomously perform behaviours in order to reach its objectives. In terms of agent control, a designer's task is therefore twofold: 1) to encode agent's behaviours in terms of actions it performs in the environment, and 2) to encode mechanisms of agent's internal deliberation leading to choices of those actions. Internal behaviours of an agent are usually deterministic and fully controlled by the agent, while, due to the dynamic and inaccessible nature of the environment, many of the exogenous actions have only partially predictable and uncertain consequences. The extent of an environment's predictability and accessibility depends on a particular application domain.

The dynamics of an environment leads to difficulties with control of an agent. Unexpected events and changes can interrupt the execution of complex behaviours, or even lead to a failure. Therefore an agent has to be able to switch its behaviours according to actual situation. Moreover, due to only partial accessibility of the environment, some situations can be indistinguishable to the agent. Therefore it is vital to allow non-deterministic choice between several potentially appropriate behaviours.

Our stance is, that in terms behaviour encoding, a developer should be able to freely mix various types of them. The selection of an appropriate behaviour should be flexible enough to accommodate a range of arbitration mechanisms, from strictly deterministic to arbitrary. These considerations roughly correspond to those for hybrid robotic architectures [1].

Knowledge representation The requirement to model an environment and to be possibly aware of own mental attitudes implies employment of some knowledge representation technology. Even though committing to a logic-based language has its advantages w.r.t. study of properties of resulting agent systems, pragmatically it is obvious that different application domains require different knowledge representation techniques. Only an application specific choice of an appropriate KR, with an associated reasoning machinery, can lead to a flexible, scalable and robust hybrid agent system. Therefore the choice of a particular KR technology should be left to an agent designer and the underlying programming framework should be modular enough to accommodate a large range of KR approaches.

Situatedness and embodiment Agents are deployed to various types of environments ranging from purely virtual to physical, or mixed. Therefore, in general, we can consider only abstract characteristics of possible environments. In the core, an agent needs means to sense its surroundings, as well as effectors to act in it.

In order not to impose arbitrary constraints on an environment, we chose a model of *active perception*, i.e. an agent actively polls it for sensory informa-

tion. W.r.t. an agent, an environment plays a rather passive role of information provider and action facilitator.

Language An agent programming language is a *glue* for assembling agent's behaviours to facilitate an efficient use of provided KR components and interface(s) to the environment. A programming language is a software engineering tool, in the first place. So even though its primary utilization is to provide expressive means for behaviour encoding, at the same time it has to fulfill high requirements on modern programming languages. Programs have to be easily readable and understandable. The semantics must be transparent, ideally with no hidden, or hardly understandable mechanisms. That makes a language easy to use and adopt, and at the same time it allows potential application of verification techniques. Finally, programs should be scalable, easily maintainable and elaboration tolerant. These requirements straightforwardly lead to a modularity and re-usability of source code, as well as strong support for program decomposition.

3 Behavioural State Machines

To tackle the desiderata discussed in the previous section, we propose *Behavioural State Machines* (*BSM*), a general purpose computational model based on the Gurevich's *Abstract State Machines* [6], adapted to the context of agent oriented programming.

The underlying abstraction is that of a transition system, similar to that used in state-of-the-art BDI agent programming languages. States are agent's mental states, collections of partial states of its knowledge bases together with a state of the environment. Transitions are induced by atomic modifiers (updates) of mental states. An agent system semantics is then a set of all enabled paths within the transition system, which the agent can traverse during its lifetime. To facilitate modularity and program decomposition, *BSM* also provide a functional view on an agent program, specifying a set of enabled transitions an agent can execute in a situation it happens to be in.

Behavioural State Machines draw a strict distinction between the *knowledge representational layer* of an agent and its *behavioural layer*. To exploit strengths of various KR technologies, the KR layer is kept abstract and *open*, so that it is possible to plug-in different KR modules as agent's knowledge bases. The main focus of *BSM* computational model is the highest level of control of an agent: its *behaviours*.

3.1 Syntax

Because of the openness of the introduced architecture, knowledge representation components of an agent are kept abstract and only their fundamental characteristics are captured by a formal definition. Basically, a KR module has to provide a language of query and update formulae and two sets of interfaces: *entailment* operators for querying the knowledge base and *update* operators to modify it.

Definition 1 ***(KR module)*** *A knowledge representation module $\mathcal{M} = (\mathcal{S}, \mathcal{L}, \mathcal{Q}, \mathcal{U})$ is characterized by*

- *a set of states $\mathcal{S}$,*
- *a knowledge representation language $\mathcal{L}$, possibly defined over some domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and variables over these domains. $\underline{\mathcal{L}} \subseteq \mathcal{L}$ denotes a fragment of $\mathcal{L}$ including only ground formulae, i.e. such that do not include variables.*
- *a set of query operators $\mathcal{Q}$. A query operator $\models \in \mathcal{Q}$ is a mapping $\models: \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \{\top, \bot\}$,*
- *a set of update operators $\mathcal{U}$. An update operator $\oplus \in \mathcal{U}$ is a mapping $\oplus : \mathcal{S} \times \underline{\mathcal{L}} \rightarrow \mathcal{S}$.*

We say that two KR languages are compatible, when they include variables over the same domain $\mathcal{D}$ and their sets of query and update operators are mutually disjoint. Two KR modules with compatible KR languages are compatible as well.

W.l.o.g., a language not including variables is compatible with any other KR language. Each query and update operator has an associated identifier. For simplicity, we do not include these in the definition, however we will use them throughout the text. When used as an identifier in a syntactic expression, we use informal prefix notation (e.g. $\models \varphi$, or $\oplus\varphi$), while when used as an operator, formally correct infix notation is used (e.g. $\sigma \models \varphi$, or $\sigma \oplus \varphi$).

Example 1 ***(running example)*** *To demonstrate the flexibility of the introduced architecture, KR technologies of our choice will be Answer Set Programming [2] and Java.*

$\mathfrak{B} = (2^{AnsProlog}, AnsProlog^*, \{\models_{ASP}\}, \{\oplus_{ASP}, \ominus_{ASP}\})$ is a KR module realizing an ASP knowledge base. The underlying language is that of $AnsProlog^*$ [2]. It includes variables over atoms and function symbols. A set of states are all well-formed $AnsProlog^*$ programs (sets of clauses). There is a single query and two update operators. Query operator $\models_{ASP}$ corresponds to skeptical version of entailment in ASP, i.e. $P \models_{ASP} \varphi$ iff φ is true in all answer sets of the program P. The two update operators $\oplus_{ASP}$ and $\ominus_{ASP}$ stand for an update by and retraction of an $AnsProlog^*$ formula (a partial program) to/from the knowledge base[2].

[2]Theoretical issues with updates of logic programs are an intensively studied research field and its deeper exploration is beyond the scope of this paper (for a basic overview see e.g. [11]).

A Java KR module $\mathfrak{C} = (\Sigma_{JavaVM}, Java, \{\models_{eval}\}, \{\oplus_{eval}\})$ *is a formalization of an interface to a running Java virtual machine. The set of Java KR module states* Σ_{JavaVM} *are all states of memory of a running VM (initialized by loading of some Java program). Both query and update operators* $\models_{eval}$, $\oplus_{eval}$ *take a Java expression and execute it in the context of the running virtual machine. Given a Java language snippet* ϕ*, the query operator* $\models_{eval}$ *returns* $\top$ *iff the expression* ϕ *evaluates to True, otherwise it returns* $\bot$*. The update operator* $\oplus_{eval}$ *simply executes a given Java expression.*

Finally, $\mathfrak{G} = (2^{Prolog}, Prolog, \{\models_{Prolog}\}, \{\oplus_{Prolog}\})$ *is a Prolog KR module, with a set of states represented as all possible sets of Prolog programs. Both, the entailment operator* $\models_{Prolog}$ *and the update operators* $\oplus_{Prolog}$ *correspond to the usual Prolog query evaluation.*

Without a formal discussion, we can say modules $\mathfrak{B}$*,* $\mathfrak{C}$ *and* $\mathfrak{G}$ *are compatible. The* $AnsProlog^*$ *language as well as Prolog include variables over terms, i.e. strings of alphanumeric characters, which is also a basic type of the Java language.*

Query formulae are the syntactical means to retrieve information from KR modules:

Definition 2 ***(query)*** *Let* $\mathcal{M}_1, \ldots, \mathcal{M}_n$ *be a set of compatible KR modules. Query formulae are inductively defined:*

- *if* $\varphi \in \mathcal{L}_i$*, and* $\models \in \mathcal{U}_i$ *corresponding to some* $\mathcal{M}_i$*, then* $\models \varphi$ *is a query formula,*
- *if* ϕ_1, ϕ_2 *are query formulae, so are* $\phi_1 \wedge \phi_2$*,* $\phi_1 \vee \phi_2$ *and* $\neg\phi_1$*.*

Query formulae can be inductively combined to compound formulae. The informal semantics is obvious: if a language expression $\varphi \in \mathcal{L}$ is evaluated to be true by a corresponding query operator $\models$ w.r.t. a state of the corresponding KR module, then $\models \varphi$ is true in that state as well.

Subsequently we define *mental state transformer*, the principal syntactic construction of *BSM* framework.

Definition 3 ***(mental state transformer)*** *Let* $\mathcal{M}_1, \ldots, \mathcal{M}_n$ *be a set of compatible KR modules. Mental state transformer expression (mst) is inductively defined:*

1. **skip** *is a mst (primitive),*
2. *if* $\oplus \in \mathcal{U}_i$ *and* $\psi \in \mathcal{L}_i$ *corresponding to some* $\mathcal{M}_i$*, then* $\oplus\psi$ *is a mst (primitive),*
3. *if* ϕ *is a query expression, and* τ *is a mst, then* $\phi \longrightarrow \tau$ *is a mst as well (conditional),*
4. *if* τ *and* τ' *are mst's, then* $\tau|\tau'$ *and* $\tau \circ \tau'$ *are mst's too (choice and sequence).*

A standalone mental state transformer is also called an *agent program* associated with a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. An update expression is a primitive mst. The other three are compound mst's (conditional, sequence and non-deterministic choice). Finally, even though we use such a notation in this paper, for brevity we omit the usual treatment of operator precedence by brackets in query formulae and mental state transformers.

Informally, a primitive mst is encoding a transition between two mental states, i.e. a primitive behaviour. Compound mst's introduce modularity and code re-use to the *BSM* framework.

Example 2 ***(running example cont.)*** *Consider a simple robot in an office environment. Modules $\mathfrak{B}$ and $\mathfrak{G}$ from the Example 1 facilitate keeping track of agent's beliefs and goals. $\mathfrak{C}$ provides an interface to the robot's physical body (an interface to its environment). Operators of the module $\mathfrak{B}$ take as an argument a formula in logical language, while those of $\mathfrak{C}$ evaluate Java expressions in a running virtual machine. Now we can write a conditional mental state transformer, demonstrating examples of both, a compound query formula and a compound mst:*

$$\models_{eval} \textit{'Visual.see("boris")'} \wedge \models_{ASP} \textit{'friend(boris)'} \longrightarrow$$

$$(\oplus_{eval} \textit{'Audio.say("Hello!")'} \circ \oplus_{ASP} \textit{'met(boris)'})$$

The mst encodes a simple behaviour applicable, when the agent encounters a person Boris, whom it considers a friend. When such a situation occurs, the robot can perform a simple ballistic behaviour consisting of first greeting Boris and subsequently remembering that this event occurred.

A mental state transformer encodes an agent behaviour. We take here a radical behaviourist viewpoint, i.e. also internal transitions are considered a behaviour. As the main task of an agent is to perform a behaviour, naturally an agent program is fully characterized by an mst and a set of associated KR modules used in it. We call such a system a *Behavioural State Machine*.

Definition 4 ***(BSM)*** *An agent system (agent) $\mathcal{A}$ is fully characterized by a* Behavioural State Machine $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$*, i.e. a collection of compatible agent KR modules and an associated agent program. We also say, that $\mathcal{A}$ is a BSM over $\mathcal{M}_1, \ldots, \mathcal{M}_n$ with the agent program $\mathcal{P}$.*

3.2 Semantics

As we sketched above, the underlying semantics of *BSM* is that of a transition system. However, before we define it rigorously, we first have to clarify the notion of state and the semantics of the involved syntactic constructs.

Definition 5 ***(state)*** *Let $\mathcal{A}$ be a BSM over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$. A state of $\mathcal{A}$ is a tuple $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of KR module states $\sigma_i \in \mathcal{S}_i$, corresponding to $\mathcal{M}_1, \ldots, \mathcal{M}_n$ respectively. $\mathfrak{S}$ denotes the space of all states over $\mathcal{A}$.*

$\sigma_1, \ldots, \sigma_n$ are partial states of the σ. State of a *BSM* corresponds to the usual notion of agent configuration (see e.g. [10]). A state can be modified by applying primitive updates on it and query formulae can be evaluated against it.

To evaluate a formula in a *BSM* state by query and update operators, the formula must be ground. However, non-ground formulae provide a more concise and practical programming style and the syntax introduced in the Subsection 3.1 explicitly allows them. Transformation of non-ground formulae to ground ones is provided by means of *variable substitution*. Variable substitution is a mapping $\theta \subseteq \{[V/T] | V \in Var_{D_i} \wedge T \in D_i \wedge 0 \leq i \leq n\}$, where $D_1, \ldots D_n$ are domains and $Var_{D_1}, \ldots, Var_{D_n}$ are sets of corresponding domain variables. By $\varphi\theta$ we denote a ground formula with all occurrences of variable $V \in Var_{D_k}$ replaced by an element $T \in D_k$, such that $[V/T] \in \theta$, for some $0 \leq k \leq n$. We consider only legal substitutions, i.e. given a KR language $\mathcal{L}$ including a domain $\mathcal{D}$, instantiation $\varphi\theta$ of a formula $\varphi \in \mathcal{L}$ is considered legal, iff $\theta \subseteq \{[V/T] | V \in Var_{\mathcal{D}} \wedge T \in \mathcal{D}\}$, i.e. $\varphi\theta \in \underline{\mathcal{L}}$.

Finally, for a compound query formula ϕ, where ϕ is $\phi_1 \wedge \phi_2$, or $\phi_1 \vee \phi_2$, or $\neg\phi_1$, formula $\phi\theta$ denotes $\phi_1\theta \wedge \phi_2\theta$, or $\phi_1\theta \vee \phi_2\theta$, or $\neg(\phi_1\theta)$ respectively. We say that variable substitution θ is *ground* w.r.t. ϕ, when $\phi\theta$ is a ground formula.

Definition 6 ***(query evaluation)*** *Let $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ be a state of a BSM with KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and ϕ be a ground query formula constructed from their corresponding KR languages. Evaluation of the query formula:*

- *when $\phi = \models_i \varphi$, where $\varphi \in \underline{\mathcal{L}_i}$ and $\models_i \in \mathcal{Q}_i$, corresponding to some $\mathcal{M}_i$, then $\sigma \models \phi$ holds, iff $\sigma_i \models_i \varphi$,*
- *when $\phi = \phi_1 \wedge \phi_2$, or $\phi = \phi_1 \vee \phi_2$, or $\phi = \neg\phi_1$, then $\sigma \models \phi$ holds, iff $(\sigma \models \phi_1) \wedge (\sigma \models \phi_2)$, or $(\sigma \models \phi_1) \vee (\sigma \models \phi_2)$, or $\sigma \not\models \phi_1$ respectively.*

Informally, a primitive ground formula is said to be true in a given *BSM* state w.r.t. a query operator, iff an execution of that operator on the state and the formula yields $\top$. For a formula from a KR language $\mathcal{L}$ of a KR module $\mathcal{M}$, only appropriate query operators are considered, i.e. those corresponding to $\mathcal{M}$.

Definition 7 ***(update and update set)*** *An update of a state σ of a BSM $\mathcal{A}$ over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ is a tuple $(\oplus, \psi)$, where $\oplus \in \mathcal{U}_i$ is an update operator of $\mathcal{M}_i$, and $\psi \in \mathcal{L}_i$ is an update formula from the KR language of that module. If ρ_1 and ρ_2 are updates, then also sequence $\rho_1 \bullet \rho_2$ is an update.*

An update set is a set of updates. A sequence of update sets $\nu_1 \bullet \nu_2$ denotes an update set $\nu = \{\rho_1 \bullet \rho_2 | (\rho_1, \rho_2) \in \nu_1 \times \nu_2\}$ iff $\nu_1 \neq \emptyset$. $\nu = \nu_2$ otherwise.

Notions of *update* and *update set* are the bearers of the semantics of mental state transformers. A simple update corresponds to a semantics of a primitive

mst, while an update set corresponds to a mst encoding a non-deterministic choice. The sequence of two update sets yields all possible sequential combinations of primitive updates from these sets. The syntactical notation of a sequence of mst's $\circ$ therefore corresponds to a sequence of updates, or update sets, denoted by the semantic sequence operator $\bullet$.

Given an update, or an update set, its application on a state of a *BSM* is straightforward. Formally:

Definition 8 ***(applying an update)*** *The result of applying an update $\rho = (\oplus, \psi)$ on a state $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ of a BSM $\mathcal{A}$ over KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ is a new state $\sigma' = \sigma \bigoplus \rho$, such that $\sigma' = \langle \sigma_1, \ldots, \sigma'_i, \ldots, \sigma_n \rangle$, where $\sigma'_i = \sigma_i \oplus \psi$, and both $\oplus \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$ correspond to some $\mathcal{M}_i$ of $\mathcal{A}$.*

Inductively, the result of applying a sequence of updates $\rho_1 \bullet \rho_2$ is a new state $\sigma'' = \sigma' \bigoplus \rho_2$, where $\sigma' = \sigma \bigoplus \rho_1$.

Mental state transformers encode *functions* yielding update sets over states of a *BSM*:

Definition 9 ***(mst semantics)*** *A mental state transformer τ of a BSM $\mathcal{A}$ yields an update set ν in a state σ under a variable substitution θ, iff $yields(\tau, \sigma, \theta, \nu)$ is derivable according to the following calculus:*

$$\frac{\top}{yields(\mathbf{skip}, \sigma, \theta, \emptyset)} \qquad \frac{\top}{yields(\oplus\psi, \sigma, \theta, \{(\oplus, \psi\theta)\})} \qquad \textit{(primitive)}$$

$$\frac{yields(\tau, \sigma, \theta, \nu), \sigma \models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \nu)} \qquad \frac{yields(\tau, \sigma, \theta, \nu), \sigma \not\models \phi\theta}{yields(\phi \longrightarrow \tau, \sigma, \theta, \emptyset)} \qquad \textit{(conditional)}$$

$$\frac{yields(\tau_1, \sigma, \theta, \nu_1), yields(\tau_2, \sigma, \theta, \nu_2)}{yields(\tau_1 | \tau_2, \sigma, \theta, \nu_1 \cup \nu_2)} \qquad \textit{(choice)}$$

$$\frac{yields(\tau_1, \sigma, \theta, \nu_1), yields(\tau_2, \sigma \bigoplus \rho, \theta, \nu_2)}{yields(\tau_1 \circ \tau_2, \sigma, \theta, \bigcup_{\rho \in \nu_1} \nu_1 \bullet \nu_2)} \qquad \textit{(sequence)}$$

The first two rules provide a semantics of primitive mst's. **skip** results in an empty update set, while a simple update yields a singleton update set. The semantics of a conditional mst is provided for both cases w.r.t. validity of the condition. If the left hand side query condition holds, the resulting update set straightforwardly corresponds to that of the right hand side mst. Otherwise, the semantics of a conditional mst is equivalent to **skip**. Conditional mst facilitates a flexible means for nesting of *BSM* code blocks.

The functional view on a mst is the primary means of compositional modularity in *BSM*. A non-deterministic choice of two, or more mst's denotes a function yielding a unification of their corresponding update sets. The result of a sequence of two, or more update sets, is a new update set consisting of sequences of updates, as defined in Definition 7. However, each subsequent update set can be only applied to a state, which is the result of applying an arbitrarily chosen update from the previous set. Given a state, a sequence of

two mst's denotes a non-deterministic choice from all possible update sets resulting from applications of different choices of an update from the first member of the sequence.

Finally, the operational semantics of an agent is defined in terms of all possible computation runs induced by a corresponding *Behavioural State Machine*.

Definition 10 ***(BSM semantics)*** *A* BSM $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ *can make a step from state σ to a state σ' (induces a transition $\sigma \rightarrow \sigma'$), if there exists a variable substitution θ, s.t. the agent program $\mathcal{P}$ yields a non-empty update set ν in σ under θ and $\sigma' = \sigma \bigoplus \rho$, where $\rho \in \nu$ is an update.*

A possibly infinite sequence of states $\sigma_1, \ldots, \sigma_i, \ldots$ is a run of BSM $\mathcal{A}$, iff for each $i \geq 1$, $\mathcal{A}$ induces a transition $\sigma_i \rightarrow \sigma_{i+1}$.

The semantics of an agent system characterized by a BSM $\mathcal{A}$, is a set of all runs of $\mathcal{A}$.

Even though the introduced semantics of *Behavioural State Machines* speaks in operational terms of sequences of mental states an agent can reach during its lifetime, the style of programming induced by the formalism of mental state transformers is rather declarative. Primitive query and update formulae are treated as black-box expressions by the introduced *BSM* formalism. On this high level of control, they rather encode *what* should be executed, while the question *how* it is done is left to the underlying KR module. I.e., *agent's deliberation abilities reside in its KR modules, while its reactive behaviours are encoded as a BSM*.

4 Agent oriented programming

The plain formalism of *Behavioural State Machines*, as introduced in Section 3, does not feature any of the usual characteristics of rational agents [20], such as *goals*, *beliefs*, *intentions, commitments* and alike, as first class citizens. As we already stated in Section 1, our position on design of an agent programming framework is rather liberal and pragmatic. It should be rather *methodological guidelines* and exploitation of degrees of freedom provided by the *BSM* programming framework, which guide a programmer to a design of a practical agent system. In the following, we discuss some of them and demonstrate how some features, desirable for rational agents, can be implemented using the *BSM* framework. However, before that we sketch an implemented instance of *BSM*.

4.1 Jazzyk

In our attempt to practically test the *BSM* approach to programming agent systems, we designed a programming language *Jazzyk* and an interpreter for

```
program ::= mst
mst ::= update | conditional | sequence | choice | '{' mst '}'
sequence ::= mst ',' mst
choice ::= mst ';' mst
conditional ::= 'when' query_expr 'then' mst ['else' mst]
query_expr ::= query 'and' query | query 'or' query |
               not 'query' | '(' query ')'
query ::= 'true' | 'false' |
          <operatorId> <moduleId> [variables] formula
update ::= 'skip' | <operatorId> <moduleId> [variables] formula
formula ::= '[{' <arbitrary string> '}]'
variables ::= '(' (<identifier> ',')* <identifier>')'
```

Figure 1: *Jazzyk* EBNF

it. Figure 1 lists the EBNF of *Jazzyk*, which straightforwardly follows from the syntax of *BSM* introduced in Subsection 3.1.

The core of *Jazzyk* syntax are rules of conditional nested mst's of the form $query \longrightarrow mst$. These are translated in *Jazzyk* as "`when <query> then <mst>`". Mst's can be joined using a sequence '`,`' and choice '`;`' operators corresponding to *BSM* operators $\circ$ and $|$ respectively. The operator precedence can be managed using braces '`{`', '`}`', resulting in an easily readable nested code blocks syntax. The query formulae are a straightforward translation of *BSM* query syntax.

Each KR module provides a set of named query and update operators, identifiers of which are used in primitive query and update expressions. To allow the interpreter to distinguish between arbitrary strings and variable identifiers in primitive query and update expressions, *Jazzyk* allows optional explicit declaration of a list of variables used in them. A standalone update expression is a shortcut for a *BSM* rule of the type $\top \rightarrow$ `<update>`. For simplicity, the EBNF in Figure 1 omits syntax for declaration and initialization of KR modules. An obvious syntactic sugar of "`when-then-else`" conditional mst is introduced as well. Moreover, *Jazzyk* implementation includes a macro language[3] enabling support for higher level code structures, like e.g. named mst's with optional arguments.

The semantics of the *Jazzyk* interpreter is that of *BSM* as described in Subsection 3.2 with few simplifications to allow for an efficient computation: 1) query expressions are evaluated sequentially from left to right, 2) the KR modules are responsible to provide a single ground variable substitution for declared free variables of a true query expression, 3) before performing an update, all the variables provided to it have to be instantiated, and 4) rules are evaluated in an arbitrary order, with the option to switch sequential evaluation on. Finally, few additional keywords like e.g. **exit** and various configuration options are introduced, together with a mechanism for initializing the initial states of KR modules.

[3] GNU M4 available at `http://www.gnu.org/software/m4/`.

4.2 BSM and AOP

Figure 2 lists an example of a *Jazzyk* code for the office space robot from Example 1. In the normal mode of operation, the robot moves randomly around and when interrupted by somebody, it smiles and utters a greeting. When it detects low battery alert, it switches off all the energy consuming features and tries to get to a docking station, where it can recharge.

In the following, we discuss the details of the robot implementation in *Jazzyk* and relate it to some desirable agent-oriented features.

Heterogeneous KR The agent uses three KR modules corresponding to those introduced in the Example 1: `beliefs` - an ASP module representing agent's beliefs ($\mathfrak{B}$ in Example 1) with a query operator `query` and two update operators `adopt`, `drop`, corresponding to $\models_{ASP}$, $\oplus_{ASP}$ and $\ominus_{ASP}$; `body` - a *Java* module for interfacing with the environment ($\mathfrak{C}$), providing query and update operators `sense` and `perform` ($\models_{eval}$ and $\oplus_{eval}$) ; and `goals` - a *Prolog* module to represent goals ($\mathfrak{G}$), with operators `query` and `update` ($\models_{Prolog}$ and $\oplus_{Prolog}$).

In the presented example, the agent uses two different knowledge bases (KB) to store information representing its mental attitudes: beliefs and goals. Instead of fixing the roles of various KBs in an agent, it is was here the programmer who chose a number and roles of KBs.

The robot employs only a single belief base to store the information about the world. However, in real-world applications it might be useful to employ several knowledge bases using heterogeneous KR technologies to facilitate a robust agent development. The *BSM* framework allows an easy integration of heterogeneous KBs in a transparent and flexible way.

Situatedness and embodiment Interaction with an environment is facilitated via the same mechanism as handling of various knowledge bases. Similarly to a KR technology, the essence of an interface to an environment are *sensor* and *effector* operators. The scheme is the same as for query and update interfaces of a pure KR module. Metaphorically, in line with behaviourist roboticists, we could say that KR module for interfacing with an environment uses the world as its own representation [1]. Moreover, given the flexibility of *BSM* framework w.r.t. heterogeneous KR technologies, an agent can easily interface with various aspects of its environment using different modules. Social and communicating agents can use specialized modules to interface with social environments they participate in (such as a FIPA compliant MAS platform) and at the same time perform actions in other environments they are embodied in.

In our example, the robot interacts with its environment via the module `body`. Figure 2 shows examples of perception handling and performing ac-

```
/* Initialization */
declare module beliefs as ASP /* initialization ommited */
declare module goals as Prolog /* initialization ommited */
declare module body as Java /* initialization ommited */

/* Perceptions handling */
when sense body (X,Y) [{ GPS.at(X,Y)}]
then adopt beliefs [{ at(X,Y) }] ;

/* Default operation */
when sense body [{ (Battery.status() == OK) }] then {
    /* Move around */
    perform body [{ Motors.turn(Rnd.get(), Rnd.get()) }] ;
    perform body [{ Motors.stepForward() }]
} else
{
    /* Safe emergency mode -- degrade gracefully */
    perform body [{ Face.smile(off) }] ;
    perform body [{ InfraEye.switch(off)}] ;
    update goals [{ assert(dock) }]
} ;

/* Goal driven behaviour */
when query goals [{ dock }] then {
        when query beliefs (X,Y) [{ position(dock_station, X, Y) }]
        then {
            perform body (X,Y) [{ Motors.turn(X,Y) }] ;
            perform body (X,Y) [{ Motors.stepForward() }]
        }
} ;

/* Commitment handling */
when query goals [{ dock }] and
      query beliefs [{ position(dock_station, X, Y), at(X,Y) }]
then update goals [{ retract(dock) }] ;

/* Interruption handling */
when sense body (X) [{ Visual.see(X) }] and
      query beliefs (X) [{ friend(X), not met(X) }]
then {
    perform body [{ Face.smile(on) }] ,
    perform body [{ Audio.say("Hello!") }] ,
    adopt beliefs (X) [{ met(X) }]
}
```

Figure 2: Example of an office space robot agent

tions.

Reactiveness and scripts The model of *Behavioural State Machines* is primarily suitable for encoding non-deterministic choice of reactive behaviours. However, it also allows encoding of script-like, or so called ballistic, behaviours [1]. An example of such is listed in Figure 2, in the part "Interruption handling", where the robot first smiles, says "*Hello!*" and finally records a notice about the event.

Such sequential, or script-like behaviours can pose a problem if an agent performs more than one exogenous action in a sequence. If the subsequent action depends on the previous one, which can possibly fail, the whole script can fail. Therefore an extensive use of script behaviours can lead to a fragile implementation. These are however considerations, which are extensively studied in robotics research [1] and we do not need to discuss them further. We only stress, that degrees of freedom, the computational model of *BSM* provides, allow programmers to exploit various programming styles and design techniques. This, however does not eradicate a need for a careful system analysis, design and consideration before using a particular technique.

Goal driven behaviour and commitment strategies The robot in our example uses goals to steer its behaviour in certain situations. Goals are used here to keep a longer-term context of the agent (docking), yet still allow it to react, change the focus to unexpected events (meeting a friend), in an agile manner.

Goals come with a certain commitment strategy. Different types of goals require different types of commitments (e.g. achievement vs. maintenance goals). The *BSM* model is quite flexible w.r.t. commitment strategy implementation. In the presented example, the commitment w.r.t. the goal $\varphi =$ `dock` can be informally written as an LTL formula $\Box(\mathfrak{G}\varphi \wedge \mathfrak{B}\varphi \rightarrow \Diamond\neg\mathfrak{G}\varphi)$ - an implementation of the axiom AI8 of the *I-System* (see Section 1).

Different commitment strategies can be implemented in the *BSM* model. The study of various types of commitment strategies and formal specification methods for *BSM* will be, however, a subject of our future work (see Section 6).

5 Discussion and related work

The primary motivation for development of the framework of *Behavioural State Machines* was 1) integration of heterogeneous knowledge representation technologies in a single agent system and 2) flexibility w.r.t. various types of behaviours. The *BSM* framework borrows the idea and the style of

KR modules integration from *Modular BDI Architecture* [13] and its theoretical foundations stem from the well studied general purpose computational model of Gurevich's *Abstract State Machines* (*ASM*) [6].

In particular, the *BSM* framework can be seen as a special class of *ASM*, with domain universes ground in the KR modules, lacking parallel execution to maintain atomicity of transition steps and featuring specialized type of sequences of updates. The close relationship to the formalism of *ASM*, allows an easy transfer of various *ASM* extensions, such as turbo, distributed, of multi-agent *ASM* [6], to *BSM* framework. Moreover, *ASM* formalism comes with a specialized modal logic for reasoning about *ASM* programs, what might turn out to be useful in the context of the already mentioned formal specification methods for *BSM*.

Modular BDI Architecture introduced decomposition of an agent system into a set of independent KR modules. However, its plain rule-based language was constrained only to BDI-type agents and did not facilitate compositional modularity w.r.t. behavioural decomposition.

To our knowledge, most implemented state-of-the-art agent oriented languages like e.g. *AgentSpeak(L)* [5], *3APL* [8], *MetateM* [9], or *Golog* [12] in their plain form are not easily extensible to handle several heterogeneous knowledge bases. The main obstacle to such an effort is a tight coupling of KR approach (usually *Prolog*, or a FOL-style language) with the language for encoding agent's behaviours (*MetateM*, *Golog*), or a tight coupling of interactions between the components of an agent system, such as a belief base and a goal base (family of languages stemming from *AgentSpeak(L)*).

GOAL [10], according to the authors, theoretically allows modularity w.r.t. the employed KR technology to some extent. However, a tight coupling of belief and goal bases via strong semantic conditions on the commitment strategy for goals imposes much stronger requirements on the integrated KR modules than the liberal approach of *BSM*. Moreover, up to our knowledge there is no published working implementation of *GOAL* with KR modularity yet.

IMPACT [17] is a system featuring a similar degree of freedom w.r.t. heterogeneous KR technologies as the *BSM* framework. It was designed to support integration of heterogeneous information sources as well as agentification of legacy applications. *IMPACT* agent consists of a set of *software packages* with a clearly defined interface comprising a set of data types, data access functions and type composition operations. An agent program is a set of declarative rules involving invocations of the underlying components via the predefined data access functions. The main difference between *IMPACT* and *BSM* framework is the semantics of agent programs. While a *BSM* program encodes a merely non-deterministic choice of conditional update expressions, and thus facilitates reactive behaviours of the agent, various semantics of *IMPACT* are strictly grounded in declarative logic programming. A set of code calls of an *IMPACT* program, to execute in each deliberation cycle, is com-

puted according to rationality requirements imposed by a particular *IMPACT* semantics. No such epistemic considerations are made for a *BSM* program. The control over which primitive mst's are enabled in a particular class of mental states is rather left to a programmer.

Finally, let us discuss how the presented *BSM* framework fits Shoham's definition of an *AOP framework* [16]. According to Shoham, a complete AOP framework will include three primary components: 1) a formal language for describing a mental state, 2) an interpreted programming language for defining agent programs faithful to the semantics of mental state, and 3) an "agentifier" converting neutral devices into programmable agents.

Obviously, we intentionally avoid to deal with the first component of an AOP framework in *BSM*. Instead we focus more on the programming language. KR language, as well as the semantics of a KR module is left open in the *BSM* framework. However, through the requirement of providing query operators, the existence of a well defined semantics of the underlying knowledge base is secured. The *BSM* programming language is merely a tool for design of a variety of interactions between the KR modules of an agent. Moreover, by adding an appropriate query/update interface to a legacy system, e.g. a relational database, it can be easily wrapped into a service agent envelope, or made a part of a more sophisticated agent system, i.e. "agentified" in the very sense of Shoham's concept of *agentification*.

6 Conclusion

The main contribution of the presented paper is the theoretical framework of *Behavioural State Machines*, an architecture for programming flexible and robust hybrid agents, exploiting heterogeneous KR technologies and allowing an easy agentification of low level interfaces and information sources. Its semantics is primarily obeying principles of *KR modularity*, and *flexibility* in terms of encoding *reactive*, as well as *sequential* behaviours. It draws a strict distinction between the *representational* vs. *behavioural* aspects of an agent and primarily focuses on the later.

We also roughly introduced *Jazzyk*, an implemented programming language for *BSM* framework. Further details with a rigorour technical description of the language will be provided in the subsequent technical note of this series.

We fall short of discussing a real world application of *Jazzyk* in this paper. In order to substantiate the presented theoretical framework in a real world experiment and drive our future research, we are currently intensively working on a showcase agent system similar to the one described in [19]: a non-trivial BDI-based bot in a simulated 3D environment of a *Quake*-based computer game. The main KR technology used for the agent's beliefs and goals will be ASP, powered by *Smodels* solver [18]. To this end we already de-

veloped several prototype *Jazzyk* KR plug-ins for *Smodels*, *Python* and a module for interaction with the *Nexuiz*[4] game server. We plan to implement a *Scheme*/*Guile*[5] module as well.

In the future, we will focus on studying higher level formal agent specification methods based on modal logic, which allow automatic translation into the plain language of *BSM*. We briefly touched on this issue in the discussion of goal oriented behaviours and commitments in Subsection 4.2. We expect this research to be pragmatically driven by the already mentioned showcase demo application.

References

[1] R. C. Arkin. *Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1998.

[2] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[3] R. H. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.

[5] R. H. Bordini, J. F. Hübner, and R. Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pages 3–37. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [4], 2005.

[6] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[7] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42(2-3):213–261, 1990.

[8] M. Dastani, M. B. van Riemsdijk, and J.-J. Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2, pages 39–68. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [4], 2005.

[4] `http://www.alientrap.org/nexuiz/`

[5] `http://www.gnu.org/software/guile/`

[9] M. Fisher. A survey of Concurrent MetateM - the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *ICTL*, volume 827 of *Lecture Notes in Computer Science*, pages 480–505. Springer, 1994.

[10] K. V. Hindriks. *Agent Programming Languages: Programming with Mental Models*. PhD thesis, Utrecht University, 2001.

[11] J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers of Artificial Intelligence and Applications*. IOS Press, 2003.

[12] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[13] P. Novák and J. Dix. Modular BDI architecture. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *AAMAS*, pages 1009–1015. ACM, 2006.

[14] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. V. de Velde and J. W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[15] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In *KR*, pages 473–484, 1991.

[16] Y. Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

[17] V. S. Subrahmanian, P. A. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[18] T. Syrjänen and I. Niemelä. The SMODELS System. In T. Eiter, W. Faber, and M. Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.

[19] M. van Lent, J. E. Laird, J. Buckman, J. Hartford, S. Houchard, K. Steinkraus, and R. Tedrake. Intelligent agents in computer games. In *AAAI/IAAI*, pages 929–930, 1999.

[20] M. Wooldridge. *Reasoning about rational agents*. MIT Press, London, 2000.